REDABAS

Arbeit mit Datenbanken (Teil I)

Dr. Thilo Weller, Matthias Donner
Karl-Marx-Universität Leipzig, Sektion Wirtschaftswissenschaften/Organisations- und Rechenzentrum

Mit dieser Beitragsfolge sollen Anregungen zur Programmierung in REDABAS vermittelt werden, die über die reine Dialogarbeit weit hinaus gehen. Gleichzeitig kommt es den Autoren darauf an, zu zeigen, daß das Standardsoftwarepaket REDABAS zahlreiche Schnittstellen zu anderen Programmen bietet, die bei der entsprechenden Aufgabenstellung ausgeschöpft werden sollten.

1. Einführung

Mit REDABAS (kompatibel zu dBASE II) steht ein universell einsetzbares RElationales DAtenBAnkbetriebsSystem für Personalcomputer/Arbeitsplatzcomputer zur Verfügung, das mit einem einheitlichen Sprachkonzept sowohl den Aufbau und die Verwaltung von Datenbeständen als auch Befehle für eine eigene Anwendungsprogrammierung bereithält /1/.
Vor allem bei komplexen Problemen kann der Bereich der Datenerfassung, Datenänderung und der Datenverknüpfung günstig mittels der integrierten Programmiersprache bearbeitet werden. Die Anwendung der REDABAS-Programme führt in den meisten Fällen zu einer Arbeitserleichterung, z. B. für Nutzer ohne rechentechnische Spezialausbildung.
Sowohl die Sprachsyntax als auch das Datenbanksystem selbst erlauben dabei eine modulare Gestaltung von Programmen, wodurch eine stufenweise Programmerstellung ermöglicht wird.

2. Syntaxnotation

Für die Notation der REDABAS-*Anweisungen* werden verwendet:
GROSSBUCHSTABEN beinhalten REDABAS-Schlüsselwörter, die genau so geschrieben werden müssen.
[...] Eckige Klammern beinhalten einen Teil der Anweisung, der wahlweise angegeben werden kann.
⟨...⟩ Kleingeschriebenes in spitzen Klammern beinhaltet einen Teil der Anweisung, der durch den Nutzer auszufüllen ist.
Für die Schreibweise der *Programmbeispiele* wird keine gesonderte Notation festgelegt.

3. Aufbau von REDABAS-Programmen

Der REDABAS-Interpreter ermöglicht also
– die Arbeit auf einer Anweisungsebene: zur Abarbeitung einmaliger Aufgabenstellungen, zum Testen von Datenbanken u. a.
– die Abarbeitung von Befehlsdateien (Programmen) mittels der integrierten Programmiersprache: mit der einfachen Möglichkeit, Anweisungen zusammenzufassen, durch einmaliges Programmieren wiederholte Nutzungsmöglichkeiten zu schaffen.
Die Programmiersprache beinhaltet zum einen die Verwendung der Anweisungen zur Datenbankarbeit auf der Anweisungsebene sowie weitere Programmstrukturelemente. Damit können z. B. solche Anweisungen auch direkt in der Programmierung genutzt werden wie zum

– Erstellen von Dateien
CREATE
COPY u. a.
– Erfassen von Daten
APPEND u. a.
– Ändern von Dateiinhalten
BROWSE
EDIT
DELETE
PACK u. a.
– Anzeigen von Dateiinhalten
DISPLAY
LIST u. a.

Zur *Erstellung* von REDABAS-Programmen kann ein Texteditor, z. B. das Textverarbeitungsprogramm TP (Option N: Bearbeiten einer Programmdatei) oder der intergrierte Programmeditor selbst genutzt werden.

4. Erstellen von Befehlsdateien mit dem integrierten Programmeditor und Aufruf von Befehlsdateien

Das *Aktivieren des Programmeditors* zur formatfreien Erstellung und Modifizierung von Anweisungsfolgen sowie zur Speicherung in einer Befehlsdatei erfolgt mit

MODIFY COMMAND ⟨programmname⟩

Nach Eingabe dieser Anweisung schaltet REDABAS in einen Eingabemodus um, wofür die vorhandenen CTRL-Tasten-Kombinationen zu verwenden sind; zum Beispiel:

CTRL – W für Speichern des Programms auf Diskette mit der Dateierweiterung. PRG (vollständiger Dateiname/max. 8 Zeichen: programmname.PRG) und Verlassen des Programmeditors
CTRL – Q für das Abbrechen der Programmbearbeitung und Verlassen des Editors *ohne* Speichern.

Zum Zeitpunkt der Programmerstellung erfolgt keinerlei Prüfung der eingegebenen Anweisungen, erst bei der Abarbeitung der Befehlsdatei!
Wichtig ist die Beachtung der Leistungsgrenzen des Programmeditors:
– maximale Länge einer Befehlszeile 77 Zeichen (Byte)
– maximale Größe einer Befehlsdatei ca. 4000 Zeichen
– keine Textsuch- sowie Blockverschiebungsanweisungen möglich.
Mittels
DO ⟨programmname⟩
kann nun der *Aufruf und die Abarbeitung* der Befehlsdatei ⟨programmname⟩. PRG erfolgen.

Die Aufrufmöglichkeiten bestehen sowohl aus der Anweisungsebene heraus (d. h. nach dem Promptzeichen „:") als auch aus einer anderen Befehlsdatei (d. h. Verschachtelungen von Befehlsdateien in Form einer Unterprogrammtechnik sind möglich). Die mögliche Schachtelungstiefe ist durch die maximale Anzahl von 16 gleichzeitig eröffneten Dateien begrenzt, wobei jede DO-Anweisung das Eröffnen einer entsprechenden (weiteren) Befehlsdatei bewirkt. Außerdem sind hier auch die anderen momentan eröffneten Dateien zu berücksichtigen (z. B. Datenbank-, Index-, Reportdateien)! Eine Befehlsdatei kann auch *direkt aus der Betriebssystemebene* heraus aufgerufen werden:

A > REDABAS ⟨programmname⟩

Die *Ausführung* einer Befehlsdatei *wird beendet*, wenn entweder das Ende der Datei oder die Anweisung

RETURN

erreicht wird.

5. Programmierte Datenein- und -ausgabe

5.1. Programmierte Dateneingabe

Die Eingabemöglichkeiten über die Konsole umfassen die
– ACCEPT-Anweisung
zur *Zeichenketteneingabe* für Speichervariable
– INPUT-Anweisung
zur Eingabe von *Daten beliebigen Typs* für Speichervariable
– READ-Anweisung
zur Eingabe von *Daten beliebigen Typs* über eine *Eingabemaske*, die zuvor mit a ... GET definiert wurde
– WAIT-Anweisung
zur Eingabe eines *Zeichens* beliebigen Typs nach einer Programmunterbrechung.
REDABAS unterscheidet also zwischen *Feldvariablen* (Diese bestimmen alle zu einer Datenbankstruktur gehörenden Datensatzelemente mit einem vom jeweiligen Datensatz abhängigen Dateninhalt.) sowie *Speichervariablen*. (Diese existieren unabhängig von der jeweils bearbeiteten Datenbank, maximale Länge jeweils 254 Byte, maximale Zahl 64.)

ACCEPT [⟨zeichenkette⟩] TO ⟨speichervariable⟩

Diese Anweisung ermöglicht die Eingabe beliebiger *alphanumerischer* Ausdrücke von der Tastatur in eine Speichervariable. Die Eingabe braucht nicht in Anführungszeichen eingeschlossen zu werden; ACCEPT faßt jede Eingabe als alphanumerischen Wert auf.
Vor der Eingabe kann eine Erläuterung (Zeichenkette) über die Art der zu erwartenden Daten gegeben werden, die in Begrenzungszeichen (Anführungszeichen ", Hochkommas ' oder eckige Klammern []) einzuschließen ist.
Die Speichervariable muß vor der Ausführung der Anweisung nicht definiert werden, sondern wird automatisch erzeugt.

Beispiele:
ACCEPT 'Programm wiederholen?' (Ja/Nein)
TO ANTWORT
ACCEPT 'Welcher Datensatz?' TO DS

Die Ausgabe der ACCEPT-Anweisung läßt sich nicht auf eine bestimmte Bildschirmposition festlegen (immer ab Spalte 0; Vorsicht: Bildschirmmenüs können leicht überschrieben werden!).

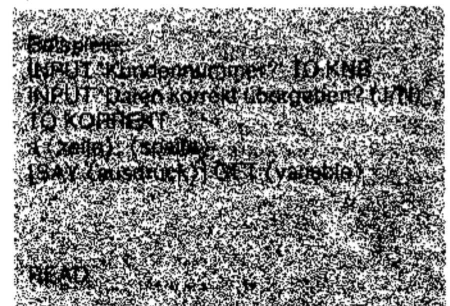
INPUT [⟨zeichenkette⟩]
TO ⟨speichervariable⟩

Die Arbeitsweise mit der INPUT-Anweisung ähnelt der bei der ACCEPT-Anweisung. Unterschiede bestehen darin, daß Daten *beliebigen* Typs eingegeben werden können; der Typ der Speichervariablen bestimmt sich durch die Art der Eingabe:

■ Numerische Werte werden numerischen Speichervariablen zugewiesen.

■ Alphanumerische Werte (hier bei der Eingabe in Begrenzungszeichen einzuschließen!) werden alphanumerischen Speichervariablen zugewiesen.

■ Bei J (oder T) bzw. N (oder F) erfolgt die Zuweisung des entsprechenden logischen Wertes TRUE bzw. FALSE an eine logische Speichervariable.


INPUT 'Kundennummer?' TO KNB
INPUT 'Daten korrekt übergeben? (J/N)'
TO KORREKT
@ ⟨zeile⟩, ⟨spalte⟩
[SAY ⟨ausdruck⟩] GET ⟨variable⟩
READ

Diese mächtigste Anweisung zum Einlesen von Daten bei der Programmabarbeitung beinhaltet die *Positionierung der Ausgabe* eines Ausdrucks auf Bildschirm (bzw. Drukker) *sowie die Eingabe von Daten beliebigen Typs* in die bezeichnete Variable.
Üblicherweise gelten folgende Positionsangaben: zeile 0...23, spalte 0...79.
Mit der SAY-Option kann ein Ausdruck (Verknüpfung von Konstanten, Variablen, Funktionen durch in REDABAS zulässige Operatoren) beliebigen Datentyps ausgegeben werden.
GET ermöglicht die Eingabe von Daten in die betreffende Variable, wobei zuerst deren momentaner Inhalt angezeigt wird.
Diese Variable *muß* vorher definiert sein (siehe auch STORE-Anweisung). Die Variable darf sowohl eine Speichervariable als auch eine (Datenbank-) Feldvariable sein.
Das eigentliche Einlesen der Daten (und deren Zuweisung zur Variablen) an den durch GET auf Bildschirm markierten Positionen erfolgt erst mit der READ-Anweisung.

Beispiel:
STORE 0,00 TO BETRAG
@ 5,10 SAY 'Betrag:' GET BETRAG
READ
Die vollständige Syntax der @...SAY-...GET-Anweisung enthält zwei weitere Optionen:

@ ⟨zeile⟩, ⟨spalte⟩
[SAY ⟨ausdruck⟩ [USING 'maske']]
[GET ⟨variable⟩ [PICTURE 'maske']]
Maske beinhaltet ein oder mehrere Maskenzeichen.

Mit der USING-Option wird festgelegt, in welcher Form die im Ausdruck stehenden Daten ausgegeben werden sollen, analog setzt die PICTURE-Option bestimmte Eingabeformate fest.

Maskenzeichen	Bedeutung für USING	Bedeutung für PICTURE
# oder 9	Ausgabe einer Ziffer des Ausdrucks (bzw. Leerzeichens, wenn an dieser Stelle keine Ziffer)	erlaubt nur Eingabe einer Ziffer sowie von "+", "-", ",", ".", " "
X	Ausgabe eines beliebigen ASCII-Zeichens	erlaubt Eingabe eines ASCII-Zeichens
A	–	erlaubt nur Eingabe eines Buchstabens
& oder *	Ausgabe einer Ziffer oder Ausgabe von "&" bzw. "*" anstelle einer führenden Null	–
!	–	Umwandlung von Klein- in Großbuchstaben

Die Standardeinstellung für die behandelte Anweisung ist SET FORMAT TO SCREEN (Bildschirmein-, ausgabe); bei Druckerausgabe ist SET FORMAT TO PRINT zu verwenden.

WAIT [TO ⟨speichervariable⟩]

WAIT veranlaßt REDABAS, das laufende Programm zu unterbrechen, seine Fortsetzung erfolgt mit der Eingabe eines beliebigen Zeichens von der Konsole. Dieses Zeichen kann wahlweise in der Speichervariablen gespeichert werden, um es für die Programmsteuerung zu nutzen.

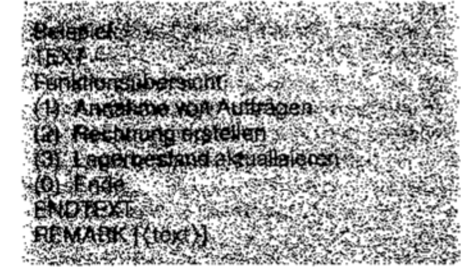


5.2. Programmierte Datenausgabe

Ausgabemöglichkeiten umfassen u. a. die
– a ... SAY-Anweisung
zur Ausgabe von Ausdrücken an bestimmten Positionen (siehe Pkt. 5.1, ohne GET-Option)
– Text ... ENDTEXT-Anweisung
zur *einfachen* Ausgabe von (z. T. umfangreichen) Kommentaren
– REMARK-Anweisung
zur weiteren Ausgabe kurzer Kommentare während der Abarbeitung von Befehlsdateien
– ERASE-Anweisung
zum Löschen des Bildschirms.

TEXT
.
.
.
⟨text⟩
.
.
.
ENDTEXT

Mit dieser Anweisungsstruktur können bei der Ausführung von Befehlsdateien eine oder mehrere Kommentarzeilen *ohne Positionsangaben* auf einfache Weise ausgegeben werden. TEXT und ENDTEXT müssen auf separaten Zeilen stehen. Der Ausgabetext erfolgt am Bildschirm *unverändert*.


TEXT
(1) Annahme von Aufträgen
(2) Rechnung erstellen
(3) Lagerbestand aktualisieren
(0) Ende
ENDTEXT
REMARK [⟨text⟩]

Die Anweisung dient der Anzeige eines (kurzen) Kommentartextes während der Programmdurchführung am Bildschirm, insbesondere zur Information des Bedieners.

ERASE

löscht den Bildschirm und positioniert den Cursor an die 1. Bildschirmposition. Wird die @-Anweisung zur Bildschirmausgabe verwendet, entaktiviert ERASE alle GET-Optionen.

5.3. Bemerkungen

Mit den im Pkt. 5 behandelten Anweisungen ist eine Vielzahl komfortabler Möglichkeiten der Online-Verarbeitung gegeben. Diese Anweisungen sollten u. a. für Menügestaltung, programmierte Datenerfassung und -aktualisierung genutzt werden.

6. Grundstrukturen der strukturierten Programmierung

6.1. Folge

Die Folge (Sequenz) beinhaltet, daß Anweisungen nacheinander abgearbeitet werden (Bild 1).
Es empfiehlt sich folgende Vorgehensweise bei der Erstellung der übergeordneten Befehlsdatei:

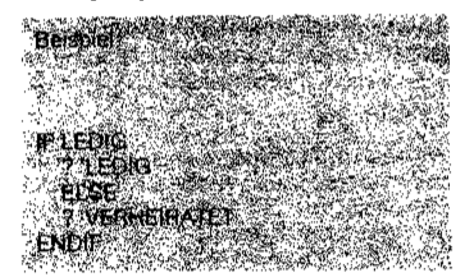
anweisung _ 1:
NOTE ⟨text⟩ bzw. *⟨text⟩ (Kommentar als Programmkopf; wird bei der Abarbeitung übergangen)
anweisung _ 2:
ERASE (Löschen des Bildschirms und Positionierung des Cursors in der linken oberen Ecke)
anweisung _ 3:
SET TALK OFF (Unterdrücken der Systemmeldungen)

6.2. Verzweigung

Die Verzweigung beinhaltet in Abhängigkeit von einer Bedingung die alternative Durchführung verschiedener Anweisungen.

6.2.1. Alternative

Mittels der IF-Anweisung wird in Abhängigkeit von der Erfüllung einer Bedingung (Ausdruck, der als Auswahlkriterium dient) die eine oder die andere Anweisung (bzw. Anweisungsfolge) ausgeführt (Bild 2).

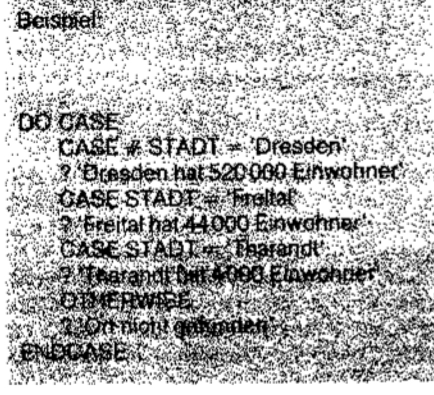

Beispiel:

```
IF LEDIG
  ? 'LEDIG'
ELSE
  ? 'VERHEIRATET'
ENDIF
```

Ist die Bedingung nach der IF-Anweisung erfüllt (wahr), werden die unmittelbar folgenden Anweisungen ausgeführt, anderenfalls die Anweisungen nach ELSE. Enthält eine IF-Anweisung keinen ELSE-Zweig (unvollständige Alternative), werden bei Nichterfüllung die Bedingung aller Anweisungen bis zur nächsten ENDIF-Anweisung übergangen.
Zwischen IF und ENDIF können z. B. weitere IF-Anweisungen stehen, die Schachtelungstiefe ist nicht begrenzt, sollte aber übersichtlich gehalten werden.

6.2.2. Fallunterscheidung

Mittels der CASE-Anweisung erfolgt die alternative Auswahl aus verschiedenen Anweisungen (Anweisungsfolgen) entsprechend den aufgeführten Bedingungen (Bild 3).

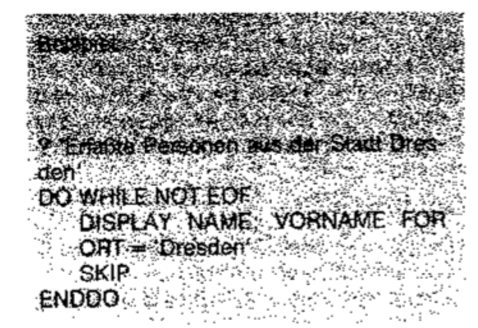

Beispiel:

```
DO CASE
   CASE # STADT = 'Dresden'
   ? 'Dresden hat 520000 Einwohner'
   CASE STADT = 'Freital'
   ? 'Freital hat 44000 Einwohner'
   CASE STADT = 'Tharandt'
   ? 'Tharandt hat 4000 Einwohner'
   OTHERWISE
   ? 'Ort nicht gefunden'
ENDCASE
```

Die Bedingungen werden nacheinander geprüft. Sobald eine Bedingung zutrifft, wird die zugehörige Anweisung bzw. Anweisungsfolge ausgeführt, und der CASE-Block ist damit abgearbeitet. Enthält CASE einen OTHERWISE-Zweig, wird, falls keine Bedingung zutrifft, die auf OTHERWISE folgende Anweisung abgearbeitet. Nach deren Abarbeiten der durch CASE bzw. OTHERWISE ausgewählten Anweisungen erfolgt die Programmfortsetzung mit der auf ENDCASE folgenden Anweisung.
Innerhalb einer Befehlsdatei dürfen CASE-Anweisungen nicht ineinander geschachtelt werden (im Gegensatz zur IF-Anweisung)!

6.3. Wiederholung

Mittels der WHILE-Anweisung wird eine Anweisung (Anweisungsfolge) abgearbeitet, solange die angeführte Bedingung erfüllt ist. Dabei wird *zuerst* geprüft, ob die Bedingung zutrifft, das heißt, gegebenenfalls wird der Block keinmal ausgeführt (Bild 4).


Beispiel:

```
? 'Erfaßte Personen aus der Stadt Dresden'
DO WHILE NOT EOF
   DISPLAY NAME, VORNAME FOR
   ORT = 'Dresden'
   SKIP
ENDDO
```

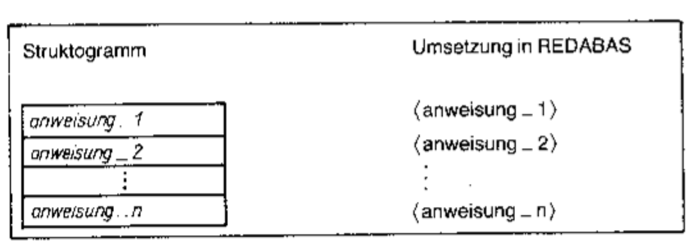
Die aufgeführte Anweisung (Anweisungsfolge) wird immer nur dann ausgeführt, wenn die Bedingung wahr ist; nach ihrer Ausführung wird die Bedingung erneut ausgewertet. DO WHILE-Verschachtelungen sind zulässig.

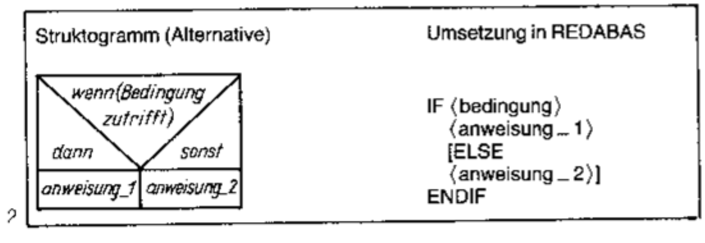
6.4. Bemerkungen

In REDABAS sind zwei wichtige Programmstrukturen *nicht* enthalten: Es fehlen die bedingte Wiederholung mit Endabfrage (Abfragebedingung am Ende), meist durch REPEAT ... UNTIL realisert, sowie die Zählschleife (n-maliges Durchlaufen einer Schleife entsprechend einem gegebenen Wert n), meist durch FOR ... TO dargestellt. Beide Strukturen müssen durch die in REDABAS vorhandenen Programmelemente ersetzt werden.

Im Sinne der strukturierten Programmierung verzichtet REDABAS auf die in vielen anderen Programmiersprachen vorhandene Sprunganweisung GOTO.

Beim Umgang mit den unter Pkt. 6.2. und 6.3. behandelten Anweisungen ist insbesondere wichtig, die richtige Stellung der jeweiligen END-Anweisungen (ENDDO, ENDIF, END-





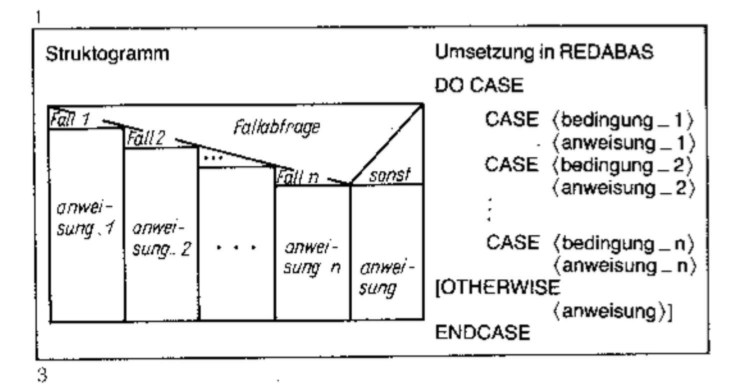


Bild 1 Sequenz von Strukturblöcken
Bild 2 Alternative
Bild 3 Fallunterscheidung
Bild 4 Abweisschleife/Kopfgesteuerte Schleife

Struktogramm | Umsetzung in REDABAS

wiederhole (solange Bedingung zutrifft) – anweisung

DO WHILE ⟨bedingung⟩
⟨anweisung⟩
ENDDO

4

CASE) zu analysieren; hier kommt bei falscher Stellung leicht die gesamte Programmlogik durcheinander!

7. Weiteres Arbeiten mit Speichervariablen

Die Verwaltung von Speichervariablen stellt insbesondere bei Befehlsdateien eine wichtige Anwendung in REDABAS dar. Da sie temporär im Arbeitsspeicher verwaltet werden, ist es aufgrund der Beschränkung auf eine maximale Zahl von 64 notwendig, beim Programmtest einen entsprechenden Überblick zu behalten:

LIST MEMORY

listet alle temporären Speichervariablen mit deren Inhalt, Größe und Typ auf.
Solche Speichervariablen bleiben solange definiert, bis REDABAS mit der QUIT-Anweisung verlassen wird (sofern sie nicht gezielt gerettet werden).
Außer den bereits behandelten Anweisungen zur Arbeit mit Speichervariablen sollen folgende noch kurz erläutert werden:

STORE ⟨ausdruck⟩ TO ⟨speichervariable⟩

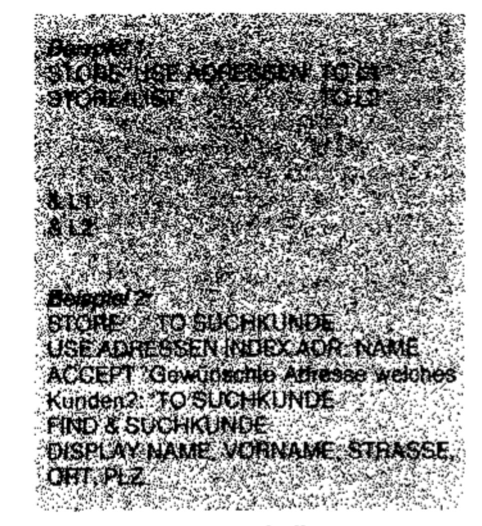
Mittels dieser Anweisung kann einer Speichervariablen der Wert eines Ausdrucks beliebigen Datentyps zugewiesen werden.
Existiert die Speichervariable noch nicht, wird sie mit dieser Anweisung angelegt. Die Zuweisung erfolgt unabhängig davon, ob die Speichervariable eventuell vorher mit einem anderen Datentyp besetzt war.

Beispiele:
```
STORE 'Montag' TO TAG
STORE 5 × 4 - 3 TO ERGEBNIS
```

Aufgrund der zahlenmäßigen Begrenzung der Speichervariablen sind weitere Anweisungen nützlich wie:
– RELEASE bzw. CLEAR
zum vollständigen/teilweisen Löschen temporärer Speichervariablen
– SAVE
zum Sichern temporärer Speichervariablen auf Diskette, die mit
– RESTORE
wieder in den Arbeitsspeicher geladen werden können.

Makroanweisung & ⟨speichervariable⟩

Die Makroanweisung in REDABAS bietet Möglichkeiten, wiederholte Eingaben z. B. von Anweisungszeilen zu vermeiden bzw. zu verkürzen sowie Anweisungen, bei denen z. B. nur Konstanten für den Suchbegriff zulässig sind, variabel zu gestalten. Sie dient damit der indirekten Adressierung von Anweisungsobjekten wie Feldern, Datensätzen, Dateien oder Anweisungen über Speichervariablen.
Trifft REDABAS in Anweisungen oder bei der Auswertung von Ausdrücken auf das Zeichen „&" (Ampersand), wird zum Zeitpunkt der Befehlsabarbeitung der in der Speichervariablen befindliche Inhalt ausgewertet. Das bedeutet, daß zur Speichervariablen zuvor eine Eingabe bzw. Zuweisung erfolgt sein muß.



Beispiel 1:
```
STORE 'USE ADRESSEN' TO L1
STORE 'LIST [illegible]' TO L2
[illegible]
& L1
& L2
```

Beispiel 2:
```
STORE ' ' TO SUCHKUNDE
USE ADRESSEN INDEX ADR_NAME
ACCEPT 'Gewünschte Adresse welches
Kunden?: ' TO SUCHKUNDE
FIND & SUCHKUNDE
DISPLAY NAME, VORNAME, STRASSE,
ORT, PLZ
```

8. Unterprogrammtechnik

Unterprogramme sind in REDABAS Teilprogramme mit gleichem Aufbau wie Befehlsdateien sowie thematisch begrenzten Anweisungsfolgen. Schachtelungen sind möglich.
Der *Aufruf* von Unterprogrammen erfolgt analog dem Aufruf von Befehlsdateien:
aus der Anweisungsebene bzw. übergeordneten Befehlsdatei heraus. Eine Parameterübergabe (Parameterliste) beim Aufruf ist *nicht* zulässig. Die *Rückkehr* in die aufrufende Ebene erfolgt mit

RETURN

Dabei wird die Abarbeitung des Unterprogramms beendet; die Programmsteuerung geht an die aufrufende Ebene zurück.

Beispiel:
```
* HAUPTPROGRAMM



DO UNTERPROGRAMM1
    DO UNTERPROGRAMM11
    DO UNTERPROGRAMM12
        DO UNTERPROGRAMM121
        DO UNTERPROGRAMM122
    DO UNTERPROGRAMM2
    DO UNTERPROGRAMM3
* HAUPTPROGRAMM – FORTSETZUNG
```

9. Hinweise zur Erstellung und zum Test von Befehlsdateien [2–4]

Die Analyse eines Anwendungsproblems hat in REDABAS eine ebenso große Bedeutung wie in anderen Programmiersprachen oder bei der Nutzung anderer Standardsoftware. Mit dem Vorteil von REDABAS, bei der Analyse/Programmierung nicht berücksichtigte Aufgaben diese später im direkten Dialog zu lösen, sollte sparsam umgegangen werden. Bediener ohne Vorkenntnisse kommen sonst kaum in die Lage, solcherart Programme effektiv zu nutzen. Damit steigen die Anforderungen an Programmsicherheit, Programmkomfort und verständlichen Programmablauf. Wichtige Gesichtspunkte bei der *Erstellung von Befehlsdateien* sind damit u. a. (siehe */2/, /3/*):

– Modulare, übersichtliche Programmierung zur Erstellung der einzelnen Arbeitsabläufe
– Prüfung des Datenvolumens
– Prüfung der Art der anzulegenden Dateien (z. B. für Stamm-, Bewegungsdaten)
– Nutzung von Bildschirmmasken
– Paßwortschutz
– optimale Gestaltung von Sortierkriterien
– Datensicherungsmaßnahmen.

Zur *Steigerung der Interpreter-Leistung* von REDABAS sind folgende allgemeine Hinweise wichtig:
Insbesondere bei mittelgroßen und großen Datenbeständen macht sich die interpretative Abarbeitung laufzeitmäßig bemerkbar.

Hier ist
1. genau der Rahmen der anzustrebenden Problemlösung abzustecken (z. B. nicht nach allen möglichen Variablen zu indizieren, die aber nur teilweise benötigt werden)
2. Programmierung aus dem Stegreif zu vermeiden,
3. nach Erstellung einer ersten Programmvariante nach weiteren, effektiveren zu suchen
4. überhaupt zu trennen, was mit REDABAS im Dialog und welche Aufgabe mittels Programmierung zu lösen ist. (Der Einsatz beider Möglichkeiten führt in den meisten Fällen zu einem optimalen Ergebnis.)

Für Programmtestläufe sind insbesondere SET-Anweisungen nützlich, wie

SET ECHO ON

zum Anzeigen aller ausgeführten Anweisungen einer Befehlsdatei während der Laufzeit auf Bildschirm (Fehlersuche!),

SET STEP ON

zur Einzelschrittabarbeitung einer Befehlsdatei mit nachfolgender Abfrage zur weiteren Abarbeitung,

SET TALK ON

zur Anzeige von REDABAS-Systemmeldungen (Voreinstellung),

SET DEBUG ON

zur Ausgabe der mit SET ECHO ON/SET STEP ON erzeugten zusätzlichen Ausgaben auf Drucker.

Einzelne Möglichkeiten zur Erhöhung der Laufzeiteffizienz sind zum Beispiel:
● bei REDABAS-Schlüsselwörtern nur die ersten 4 Zeichen zu verwenden (reicht dem Interpreter zur Identifizierung):

aus	wird
CREATE	CREA
DISPLAY	DISP

● die Verwendung der LOOP-Anweisung in DO WHILE-Schleifen zur sofortigen Rückkehr bei Abbruchbedingungen an den Schleifenanfang, ohne die Anweisungen bis zum nächsten ENDDO erst auszuführen (Zeitersparnis)
● statt WAIT die ACCEPT-Anweisung einzugeben (Bei WAIT erfolgt Nachladen von Diskette)

wird fortgesetzt

REDABAS

Arbeit mit Datenbanken (Teil II)

Dr. Thilo Weller, Matthias Donner
Karl-Marx-Universität Leipzig,
Sektion Wirtschaftswissenschaften/
Organisations- und Rechenzentrum

• Diskettenzugriffe, insbesondere beim indizierten Zugriff: Wenn die Datenbank oder/und die Index-Dateien durch häufige Datenergänzung und -modifikation auf dem externen Speicher nicht mehr in hintereinanderliegenden Blöcken verteilt ist bzw. sind, verzögern sich Diskettenzugriffe.
Dies läßt sich dadurch beseitigen, daß die Diskette unter Betriebssystemkontrolle mittels der PIP-Kommandos unter SCP kopiert wird, wobei getrennt abgelegte, zu einer Datei gehörende Blöcke wieder direkt hintereinander gespeichert werden und die Zugriffszeiten bei REDABAS sich verkürzen.
• Die Zugriffszeit beim Arbeiten mit indexsequentiellen Zugriffen wird weiter optimiert, wenn die Datenbank selbst in der Reihenfolge vorsortiert ist, wie es sonst durch den Indexhauptschlüssel geschieht. Vor allem bei Daten geringer Fluktuation (Stamm-Daten) sollte dies beachtet werden. Die Sortierung läßt sich leicht erreichen, indem die Datenbank mit der gewünschten Index-Datei aktiviert und dann in eine neue Datei kopiert wird.
Anschließend kann z. B. die bisherige Datenbank gelöscht, die neue umbenannt und mit dieser auch ohne Index-Datei weitergearbeitet werden.
• Die Nutzung der COPY-Anweisung unter REDABAS sollte nur bei entsprechend verwendeten Optionen wir Feldauswahl u. ä. eingesetzt werden; sie ist wesentlich zeitaufwendiger als z. B. PIP unter SCP.

10. Schnittstelle von REDABAS zu weiterer Software

Für die Kommunikation zwischen REDABAS und anderer Standardsoftware (z. B. Textverarbeitungsprogramm TP, Kalkulationsprogramm KP) bzw. Anwenderprogrammen, z. B. BASIC oder PASCAL, steht eine Schnittstelle auf Datenebene zur Verfügung:

Standardsoftware bzw. Anwenderprogramme ←SDF→ REDABAS

REDABAS ist mit diesem SDF (System-Daten-Format; ASCII-Standard-Datenformat) in der Lage,
– Datenbankkopien in einem Format zu erzeugen, die auch von anderen Anwenderprogrammen oder anderer Standardsoftware verarbeitet werden können
– Dateien selbst zu verarbeiten, die durch solche Programme erstellt werden.
Der Typ dieser Dateien ist eine Textdatei, deren Dateninhalt aus beliebigen Zeichen des ASCII-Codes besteht und jeder Datensatz mit Wagenrücklauf und Zeilenvorschub (CR/LF) abgeschlossen ist. Die Datensätze sind sequentiell angeordnet und können konstant oder variabel sein.

10.1. Bereitstellung von REDABAS-Datenbanken

COPY TO ‹dateiname› [SDF]
[DELIMITED] [WITH ‹trennzeichen›]

REDABAS erzeugt damit eine Kopie der aktivierten Datenbank vom Typ Textdatei (dateiname. TXT) im ASCII-Standard-Datenformat. Das Ausgabeformat der Sätze hängt davon ab, welche Optionen in der COPY-Anweisung verwendet werden:

– COPY TO ‹dateiname› SDF

Die Felder werden in voller, definierter Länge *ohne* Begrenzungs- und Trennzeichen in der Textdatei angeordnet. Numerische Werte werden innerhalb der Felder rechtsbündig, Zeichenketten linksbündig dargestellt.

– COPY TO ‹dateiname› DELIMITED

Alle Felder im Ausgabesatz sind durch Kommas voneinander getrennt. Zeichenketten sind in Hochkommas eingeschlossen, endständige Leerstellen werden entfernt. DELIMITED schließt SDF automatisch ein.

– COPY TO ‹dateiname› DELIMITED WITH ‹trennzeichen›

Führende Leerstellen in numerischen Feldern und endständige Leerstellen bei Zeichenkettenfeldern werden entfernt. Die Felder sind durch das in der WITH-Option angegebene (d. h. dem WITH unmittelbar folgende) Trennzeichen (Sonderzeichen) voneinander getrennt und enthalten keine zusätzlichen Begrenzungszeichen.

Soll von der Ausgangsdatenbank nur ein Teil (z. B. ausgewählte Feldnamen, Datensätze) in die Textdatei übernommen werden, ist die COPY-Anweisung mit den entsprechenden Optionen zu nutzen:

```
COPY TO ‹dateiname› [‹bereich›]
          [FIELD ‹feldnamen›]
        [FOR ‹bedingung›] [SDF]
          [DELIMITED
          [WITH ‹trennzeichen›]]
```

10.2. Übernahme von Dateien in REDABAS

APPEND FROM ‹dateiname› [FOR ‹bedingung›] [SDF] [DELIMITED]

REDABAS verarbeitet Textdateien, die von anderen Programmen erzeugt wurden, mit der erweiterten APPEND-Anweisung. Dabei werden die Datensätze aus der Datei dateiname (im ASCII-Standard-Datenformat) in die aktivierte Datenbank übernommen. Die Reihenfolge der Felder in der Textdatei muß der definierten Feldanordnung in der aktiven Datenbankdatei entsprechen, an die die Datensätze der Textdatei angefügt werden.
Analog zur COPY-Anweisung können die Datenfelder unterschiedlich begrenzt sein:
– APPEND FROM ‹dateiname›
  [FOR ‹bedingung›] SDF

Die Felder in der Textdatei werden in der gleichen Länge erwartet, wie sie in der aktiven Datenbankdatei definiert sind. Die Datenfelder müssen in der Textdatei ohne Trenn- und Begrenzungszeichen angeordnet sein.

– APPEND FROM ‹dateiname›
  [FOR ‹bedingung›] DELIMITED

REDABAS erwartet die Datenfelder in der Textdatei durch Kommas voneinander getrennt. Felder vom Typ Zeichenkette können in Hochkommas oder Anführungszeichen eingeschlossen sein. Andere Begrenzungszeichen werden zum Feldinhalt gehörig betrachtet.

10.3. Beispiel Schnittstelle REDABAS – TP

Mittels des Textprogramms TP soll ein Serienbrief erstellt werden, der für jeden Empfänger passend gestaltet ist.
Die entsprechenden Angaben des Empfängers sollen einer REDABAS-Datei ADRESSEN.DBD folgender Struktur entnommen werden.

FELD	NAME	TYP	LÄNGE	DEZ
001	NAME	C	010	
002	VORNAME	C	008	
003	STRASSE	C	015	
004	ORT	C	008	
005	PLZ	N	004	

1. Schritt

Konvertierung (Kopieren) der REDABAS-Datei ADRESSEN.DBD in eine Textdatei ADRESSEN.TXT mit Kommas als Trennzeichen (TP verarbeitet als Trennzeichen zwischen Datenfeldern Kommas).
USE ADRESSEN
COPY TO ADRESSEN DELIMITED WITH,

2. Schritt

Erstellen einer Textschablone mit TP
VEB Gebäudewirtschaft Dresden
· op (Ausschalten der Seitenwechselanzeige)
· df B: ADRESSEN.TXT
(Angabe der entsprechenden Datei, hier im Laufwerk B)
· rv NAME, VORNAME,
STRASSE, ORT, PLZ
(Angabe der Feldnamen; Übereinstimmung mit der Ausgangsdatenbank bzgl. Reihenfolge beachten!)
Familie
& NAME & & VORNAME & (Markierung

```
& STRASSE &        der Textstel-
& ORT &            len, die später
& PLZ &            durch konkrete
                   Werte ersetzt
                   werden)
                Dresden, 30. März 1987
Werte Familie & Name &!
Wir möchten Sie bitten, zur Überreichung des
Mietvertrages . . .
. . .
. . .
```

3. Schritt

Ausdruck der Serienbriefe mit TP (Option M: Kombo-Druck)

Beim Ausdruck wird jeder Variablenname durch einen Variablenwert (aus der Datei ADRESSEN.TXT) ersetzt:

```
VEB Gebäudewirtschaft Dresden
Familie
Lehmann Kurt
Hauptstr. 15
Dresden
8060
                Dresden, 30. März 1987
Werte Familie Lehmann!
. . .
```

10.4. Schnittstelle REDABAS – Höhere Programmiersprache

Für diese Schnittstelle wird als Beispiel die Programmiersprache PASCAL verwendet (System PASCAL 880/S /5/, Sprache kompatibel zu TURBO-PASCAL).
Die Schnittstelle kann nützlich sein, um bestimmte Teilaufgaben (z.B. Sortierung) in dieser höheren Programmiersprache und damit wesentlich schneller zu realisieren. Als Ausgangspunkt dient wiederum die Datenbank ADRESSEN.DBD mit analoger Struktur (siehe 10.3.). Ziel soll es sein, die Datensätze dieser Datei unter Pascal verfügbar zu machen.

1. Schritt

Konvertieren (Kopieren von ADRESSEN.DBD in eine Textdatei ADR.TXT im SDF-Format.
USE ADRESSEN COPY TO ADR SDF

2. Schritt

Erstellen eines PASCAL-Programms zur Übernahme dieser Datensätze aus der temporären Datei ADR.TXT in die Datei ADR.DAT

Thilo Weller ist Hochschuldozent an der Sektion Wirtschaftswissenschaften der Karl-Marx-Universität Leipzig. Nach dem Studium der Physik an der KMU 1967–1974 arbeitete er bis zur Promotion B auf dem Gebiet rechenintensiver molekülphysikalischer Untersuchungen. Mit der breiteren Entwicklung der Mikrorechentechnik wandte er sich dann Vorlaufuntersuchungen zu dieser Problematik in gesellschaftswissenschaftlichen Fragestellungen zu und befaßt sich derzeit im Rahmen der Wirtschaftsinformatik mit Fragen des Hard- und Softwareeinsatzes vernetzter Arbeitsplatzrechner.

Matthias Donner ist wissenschaftlicher Assistent am Organisations- und Rechenzentrum der Karl-Marx-Universität Leipzig. Er studierte von 1977–1981 an der Technischen Hochschule Leipzig Technische Kybernetik und Automatisierungstechnik.
In seiner Tätigkeit am ORZ der KMU widmet er sich insbesondere dem Problemkreis der Mikrorechentechnik, speziell deren Nutzung zur digitalen Simulation.

```
program REDABAS_TURBO_DATEI_CONVERTIERUNG
type                     {Vereinbarung spezi-
                          fischer Datentypen}
  Adresse = record
        Name    : string [10];
        Vorname: string [8];
        Straße  : string [15];
        Ort     : string [8];
    end;
AdressDatei = File of Adresse;
var
  Datensatz: Adresse
             {Vereinbarung von Programm-
              variablen}
  Datei      : AdressDatei;
  TF         : text;
begin
  ClrScr;
  assign(TF, 'ADR.TXT'); reset(TF);
                  {Aktivieren von ADR.TXT zum
                   Lesen}
  assign(Datei, 'ADR.DAT'); rewrite(Datei);
                  {Generierung von ADR.DAT}
  while not EOF(TF) do
    begin
      with Datensatz do
        begin
          read(TF, Name);
                      {Lesen der einzelnen
                       Datensatz-Felder}
          read(TF, Vorname);
          read(TF, Straße);
          read(TF, Ort);
          read(TF, PLZ)
                   {Letztes Feld mit readln
                    lesen}
        end;
      write(Datei, Datensatz);
                    {Schreiben des Datensat-
                     zes in ADR.DAT}
      write('–',Datensatz,Name)
                    {Protokoll}
    end;
  close(Datei); {Schließen der Adreßdatei
                 ADR.DAT}
  close(TF);    {Schließen der Textdatei
                 ADR.TXT}
end.
```

3. Schritt

Weiterbearbeitung der REDABAS-Adreßdatei unter PASCAL 880/S.

10.5. Schnittstelle höhere Programmiersprache – REDABAS

Als Beispiel soll eine unter PASCAL 880/S bearbeitete Adreßdatei ADR.1DAT für REDABAS verfügbar gemacht werden.

1. Schritt

Entwickeln eines PASCAL-Programms zum Kopieren der Datensätze von ADR.1.DAT in eine temporäre Textdatei ADR1.TXT (Die Programmversion soll *eine* Möglichkeit aufzeigen).

```
program TURBO_REDBAS_DATEI_CONVERTIERUNG;
type                     {Vereinbarung spezi-
                          fischer Datentypen}
  Adresse = record {Datensatz-Vereinba-
                    rung}
    Name    : string [10];
    Vorname: string [8];
    Straße  : string [15];
    Ort     : string [8];
    PLZ     : string [4];
    end;
  Adress-Datei = File of Adresse;
var                {Vereinbarung von Pro-
                    grammvariablen}
  Datensatz: Adresse;
  Datei     : AdressDatei;
  TF        : text;
  i         : integer;

begin
  ClrScr;
  assign(Datei, 'ADR1.DAT'); reset(Datei);
                  {Aktivieren von
                   ADR1.DAT zum Lesen}
  assign(TF, 'ADR1.TXT'); rewrite(TF);
                  {Generierung von
                   ADR1.TXT}
  while not EOF(Datei) do
    begin
      read(Datei, Datensatz);
                  {Lesen eines Daten-
                   satzes aus ADR1.DAT}
      with Datensatz do
      begin        {Schreiben der Daten-
                    satzfelder}
      write (TF,Name);
      if length(Name) < 10 then
        for i: = length(Name)+1 to 10
            do write(TF,' '); {Auffüllen
                 mit Leerzeichen}
      write(TF, Vorname);
      if length(Vorname) < 8 then
        for i: = length(Vorname)+1 to 8
            do write(TF, ' ');
      write(TF, Straße);
      if length(Straße) < 15 then
        for i: = length(Straße)+1 to 15
            do write(TF,' ');
      write(TF, Ort)
      if length(Ort) < 8 then
        for i: = length(Ort)+1 to 8
        do write(TF, ' ');
      writeln(TF,PLZ);
    end;
    write('–',Datensatz.Name);
                  {Protokoll}
  end;
  close(Datei);  {Schließen der Adreßdatei
                  ADR1.DAT}
  close(TF);     {Schließen der Textdatei
                  ADR1.TXT}
end.
```

2. Schritt

Kopieren der Struktur von ADRESSEN.DBD nach ADR1.DBD (in REDABAS)
USE ADRESSEN
COPY STRUCTURE TO ADR1

3. Schritt

Anfügen der Datensätze aus der temporären Textdatei ADR1.TXT an ADR1.DBD
USE ADR1
APPEND FROM ADR1.TXT SDF

Fortsetzung auf Seite 19

Fortsetzung von Seite 14

11. Der interne Aufbau einer Datenbank

Für die Untersuchung des internen Aufbaus einer beliebigen Datei – also auch einer Datenbank – können Debugger wie DU oder andere Monitorprogramme genutzt werden. Eine solche Betrachtung gibt Auskunft über die REDABAS-interne Verwaltung der Strukturvereinbarungen (Tafel 1).

Tafel 1

← REDABAS – Datenbank →			
Struktur-vereinbarung	1. Datensatz	...	n. Datensatz

Byte 0 – 519

REDABAS reserviert die ersten 520 Bytes für die Beschreibung der Datenbank bzw. der Datenfelder. Im ersten Byte (Byte 0) steht mit 02h ein Wert, der wohl vordergründig aus Kompatibilitätsgründen seine Daseinsberechtigung hat. Die nächsten zwei Bytes enthalten die Gesamtzahl der Datensätze, wozu anzumerken ist, daß das niederwertige *vor* dem höherwertigen Byte gespeichert ist. In den nächsten drei Bytes steht das Datum der letzten Änderung der Datenbank in der Form TT/MM/JJ (hexadezimal). Die um eins erhöhte Länge eines Datensatzes ist in den Bytes 6 und 7 gespeichert. Soweit die allgemeinen Angaben.
Die restlichen Bytes (Byte 8 bis 519) sind für die Beschreibung der n Datenfelder (1 < = n < = 32) vorgesehen (Tafel 2).

Tafel 2

Byte	Wert	Bedeutung
00	02h	
01–02	xx xx	Anzahl der Datensätze
03–05	xx xx xx	Datum (TT MM JJ)
06–07	xx xx	Datensatzlänge + 1
08–519		Strukturdefinitionen
520–n	20h	Beginn Datenbereich (1. Datensatz)
n + 1	1Ah	Dateiende

Zur Feldbeschreibung dient jeweils ein 16-Byte-Bereich, der alle zur eindeutigen Festlegung der Struktur erforderlichen Definitionen enthält (Tafel 3).

Tafel 3

Byte	Wert	Bedeutung
00–09		Feldname
10	00h	
11	xx	Feldtyp
12	xx	Feldlänge
13–14		reserviert
15	xx	Anzahl Dezimalstellen

Ein 0Dh im Byte 0 des 16-Byte-Bereichs kennzeichnet den Abschluß der Strukturdefinitionen.
Der Datenbereich beginnt mit dem Byte 520.
Am Anfang eines jeden Datensatzes steht ein Leerzeichen (20h), dem sich die Daten entsprechend der Strukturvereinbarung anschließen. Abgeschlossen wird die Datenbank mit dem Dateiendekennzeichen 1Ah.
Das nachfolgende PASCAL-Programm analysiert die Struktur einer beliebigen REDABAS-Datenbank und bringt diese auf dem Bildschirm zur Anzeige.
Dieses einfache PASCAL-Demonstrations-Programm zeigt die Möglichkeit der Weiterverarbeitung von REDABAS-Datenbank-Informationen unter Nutzung einer höheren Programmiersprache auf.

```
Program Redabas_Datenbank_Struktur;
const
  alphanum : set of char =
                    Ä'A'..'Z','_','0'..'9'ü;
type
  eintrag = record
              name: stringÄ16ü;
              laenge: byte;
              typ: (C, N, L)
            end;
  zeile = stringÄ80ü;
  bytearray = arrayÄ0..520ü of byte;
var
  quelle,ziel : file;
  puffer : bytearray;
  dateistruktur : record
              eintrzahl:integer;
              eintr:arrayÄO..127ü
                      of eintrag
            end;
  quellname,zielname : stringÄ30ü;
  gesamtzahl,satzlänge, anzahlptr,
  anfangptr,offset,längenoffset,
  typoffset : integer;
  Redabas_Byte_0 : byte;
procedure fehlernachricht (meldung:zeile);
begin writeln(meldung); halt end;
procedure analyse;
var
  i, j:integer;
  lowbyte,highbyte,typflag,letter:byte;
  begin (* analyse *)
    blockread(quelle,puffer,4);
(* liest die ersten 512 Byte der Datei *)
    Redabas_Byte_0:=pufferÄ0ü
    if Redabas_Byte_0=2 then
  begin
    anzahlptr:=0;anfangptr:=8;
    offset:=16;typoffset:=11;
    längenoffset:=12
     end
    else fehlernachricht
                ('Keine Redabas-Datei');
    lowbyte:=pufferÄ1ü;
    highbyte:=pufferÄ2ü;
    gesamtzahl:=(highbyte shl 8)
                      or lowbyte;
    lowbyte:=pufferÄ6ü;
    highbyte:=pufferÄ7ü;
    satzlänge:=(highbyte shl 8)
                  or lowbyte;
    dateistruktur.eintrzahl:=0;
    writeln(Gesamtzahl, 'Einträge a',
          satzlänge, 'Byte.');
    writeln;
    writeln('Feld                  Länge Typ');
    writeln;
    with dateistruktur do
      repeat
        i:=(eintrzahl*offset)+anfangptr;
        j:=i;
        letter:=pufferÄiü;
        if letter <>$0D then
        with eintrÄsucc (eintrzahl)ü do
          begin
            name:=chr (letter);
            repeat
              j:=succ (j);
              letter:=pufferÄjü;
              if chr (letter)
               in alphanum then
               name:=name+chr(letter)
            until not
              (chr(letter) in alphanum);
            typflag:=puffer
                    Äi+typoffsetü;
            länge:=puffer
                    Äi+längenoffsetü;
            write(name,
                 '':12-length(name),
                 länge:4,
                 chr(typflag):4);
            case chr(typflag) of
              'C' : typ:=C;
              'N' : typ:=N;
              'L' : typ:=L;
             else typ:=C
            end;
            eintrzahl:=succ(eintrzahl);
            writeln
          end
      until letter=$0D;
    writeln
  end;    (*  analyse  *)

BEGIN
  writeln
  ('REDABAS – Datenbank-Informationen');
  writeln; write('Datei :');
  readln(quellname);
  if pos('.',quellname)=0 then
    quellname:=quellname+'.DBD';
  assign(quelle, quellname);
  (*I–*)
  (* schaltet Fehlerbehandlung *)
  (* durch das System aus *)
  reset(quelle);
  if ioresult<>0 then
    fehlernachricht
    ('Keine gültige Datei?');
  analyse;
  close(quelle)
END.
```

Schluß

Literatur

/1/ REDABAS Programmbeschreibung, Systemunterlagendokumentation. VEB Robotron-Projekt, Dresden, 1985
/2/ Eggerichs, W.: dBASE II (Bd. 1 und 2). VEB Verlag Technik, Berlin, 1986
/3/ Eggerichs, W.: dBASE II. Band 3: Aufbau und Nutzung von Datenbanken. Dr. A. Hüthig Verlag, Heidelberg 1986
/4/ Donner, M; Weller, Th.: Hochschul-Folien/-Dia-Reihe Datenbanksystem REDABAS. HFR 837/858/872, HR 1494/1515/1529. Institut für Film, Bild und Ton, Berlin, 1987
/5/ Bedienungsanleitung und Sprachbeschreibung für das Programmiersystem PASCAL 880/S unter Steuerung des Betriebssystems SCPX, Systemunterlagendokumentation. VEB Robotron Büromaschinenwerk, Sömmerda, 1987

Erkennung von Eingabefehlern in REDABAS-Programmen

Oskar Schönherr, Karl-Marx-Stadt

Beim Einsatz von Programmen ist es insbesondere notwendig, eine störungsfreie Arbeit auch dem Nutzer zu garantieren, der das Programm selbst nicht erstellt hat und dessen inneren Aufbau nicht kennt. Es muß erreicht werden, daß möglichst viele Fehleingaben vom Programm selbst erkannt werden. Bei der Arbeit mit REDABAS wurden dazu vom Autor eine Reihe besonders für Einsteiger geeignete Programmroutinen verwendet, die auf Grund ihrer universellen Anwendbarkeit und ihres einfachen Aufbaus für viele Anwendungsfälle leicht nachzuvollziehen sind. Es wurde davon ausgegangen, daß die Fehlererkennung bereits bei der Eingabe erfolgen sollte und nicht erst in der weiteren Programmabarbeitung, um Rücksprünge in vorangegangene Programmteile zu vermeiden. Günstig für die Bearbeitung war die Tatsache, daß sich mit REDABAS alle Programme in Menütechnik aufbauen lassen. Im folgenden werden die in den Bildern 1 bis 3 dargestellten Programmteile kurz in ihrer Wirkung beschrieben. Die einzugebende Zeichenvariable in den Beispielen heißt zvar.

1. Auswahl aus einem Menü

Wenn aus einem Menü eine Funktion ausgewählt werden soll, garantiert die im Bild 1 vorgestellte Form, daß nur die Möglichkeiten, die das Menü vorsieht, auch als echte Eingaben angenommen werden. Alle anderen Eingaben erscheinen auf dem Bildschirm, führen aber nicht zum Verlassen der Schleife. Bei einer Erweiterung des Menüs kann durch eine Veränderung der Zeichenkette die DO-WHILE-Schleife schnell auf die neuen Bedingungen angepaßt werden.

```
C:BEISP1   .PRG

*
* Auswahl aus einem Menue
*
ERASE
@  6,20 SAY '1 - Eingabe'
@  7,20 SAY '2 - Aenderung'
@  8,20 SAY '3 - Recherche'
@  9,20 SAY '4 - Austragen'
@ 10,20 SAY 'E - Ende'
@ 12,30 SAY 'Auswahl'
STORE ' ' TO zvar
DO WHILE .NOT. zvar$'1234E'
  @ 12,40 GET zvar
  READ
ENDDO
*
* Weiter mit Fallunterscheidungen
*
```

Bild 1 Funktionsauswahl

```
CO=

C:BEISP2   .PRG

*
* Auswahl laengerer Zeichenketten
*
SET TALK OFF
ERASE
@  6,20 SAY 'Auswahl der Berufsgruppe '
@  8,30 SAY '1 - Lehrling'
@  9,30 SAY '2 - Arbeiter'
@ 10,30 SAY '3 - Angestellter'
STORE ' ' TO zvar
DO WHILE .NOT. zvar$'123'
   STORE '2' TO zvar
   @ 6,45 GET zvar
   READ
ENDDO
DO CASE
   CASE zvar='1'
      STORE 'Lehrling' TO zvar
      @ 6,46 SAY zvar+CHR(20)
   CASE zvar='2'
      STORE 'Arbeiter' TO zvar
      @ 6,46 SAY zvar+CHR(20)
   CASE zvar='3'
      STORE 'Angestellter' TO zvar
      @ 6,46 SAY zvar+CHR(20) ,
ENDCASE
*
* Weiter im Programm
*
```

Bild 2 Hilfsmenü für längere Zeichenketten

2. Auswahl längerer Zeichenketten durch ein Hilfsmenü

Oft besteht die Notwendigkeit, längere Zeichenketten einzugeben, die immer wieder auftauchen. Dabei ist eine garantiert gleiche Schreibweise gefordert, wenn nach diesen Zeichenketten in Dateien recherchiert werden soll. Bewährt hat sich dabei die in Bild 2 vorgestellte Variante, die in einer angenommenen Datei mit Berufsgruppen die Recherche mit immer gleichen Zeichenketten (Frage der Schreibweise bei verschiedenen Anwendern bzw. Verhinderung von Eingabefehlern) gewährleistet.
Zur Bedienungsfreundlichkeit wurde die am meisten verwendete Berufsgruppe vorgewählt und kann mit ET angefordert werden. Zusätzlich wird deutlich, daß es damit auch einem Unkundigen möglich ist, das Programm abzuarbeiten, ohne Fehler zu provozieren. Als Hilfe wird die ausgewählte Eingabe angezeigt.

3. Auswahl aus größeren Zeichenkettenmengen

Es ist in vielen Fällen nötig, daß Zeichenketten unterschiedlichster Länge aus einem Datenbestand (aus einem Feld) herausgesucht und für die weitere Bearbeitung diese zutreffenden Datensätze zur Verfügung gestellt werden müssen. Ein Beispiel dieser Art ist die Suche nach einem Namen in einer Adreßdatei. Bei Indizierung dieser Datei nach Familienname ('name') und Vorname ('vorname') muß zum Finden der Person der Name vollständig und in der Schreibweise richtig eingegeben werden.
Das im Bild 3 dargestellte Prinzipbeispiel findet in begrenzt großen Dateien auf schnelle Art den richtigen Namen, ohne daß dieser komplett eingegeben werden muß. Ähnlich funktioniert zum Beispiel die Eingabe von Reisezielen in den Schaltergeräten der Deutschen Reichsbahn.
Das vorgestellte Prinzip hat seine Grenzen in der mit Datensatzanzahl steigenden Laufzeit und bei sehr vielen gleichen Namen. Für viele Probleme ist es aber sofort anwendbar.

```
CO=

C:BEISP3   .PRG

*
* Auswahl aus groeszeren Zeichenkettenmengen (Dateien)
*
ERAS
USE c:namen INDE c:namen
SET EXAC OFF
SET TALK OFF
SET COLO OFF
SET CONF OFF
STOR ' ' TO name1,name2
STOR 30 TO A
STOR 'X' TO zvar
@ 6,10 SAY 'Versuch Schnelleingabeverfahren'
@ 7,15 SAY 'Buchstaben- oder <ET>-Taste  => weiter mit Namen
@ 8,15 SAY 'Leertaste                    => Name richtig'
@ 10,10 SAY 'Namenseingabe :'
@ 10,30 GET name2 PICT '!'
READ
CLEA GETS
FIND &name2
IF #=0
STOR ' ' TO zvar
@ 10,30 SAY 'Gesuchter Name fehlt'+CHR(20)
ELSE
@ 10,A SAY name
@ 11,A SAY vorname
STOR A+1 TO A
DO WHIL zvar#' '
STOR $(name,A-29,1) TO name1
@ 10,A GET name1 PICT '!'
READ
CLEA GETS
IF name1<>' '
SKIP
IF name2#$(name,1,A-30)
STOR ' ' TO zvar
@ 10,30 SAY 'Gesuchter Name fehlt'+CHR(20)
LOOP
ELSE
STOR name2+name1 TO name2
STOR A+1 TO A
DO WHIL name2#$(name,1,A-30).AND..NOT.EOF
SKIP
ENDD
@ 10,30 SAY name
@ 11,30 SAY vorname
LOOP
ENDI
ELSE
STOR name TO name2
@ 10,30 SAY name
@ 11,30 SAY vorname
STOR ' ' TO zvar
ENDI
ENDD zvar#'
ENDI #=0
*
* Weiter im Programm
*
* Verwendete Datei   namen.dbd
*     Feld 1     name       C  020
*     Feld 2     vorname    C  020
*     Feld 3     strasze    C  040
*     Feld 4     ort        C  040
*     usw.
*
* INDEX ON name+vorname TO namen
```

Bild 3 Auswahl aus größeren Zeichenkettenmengen